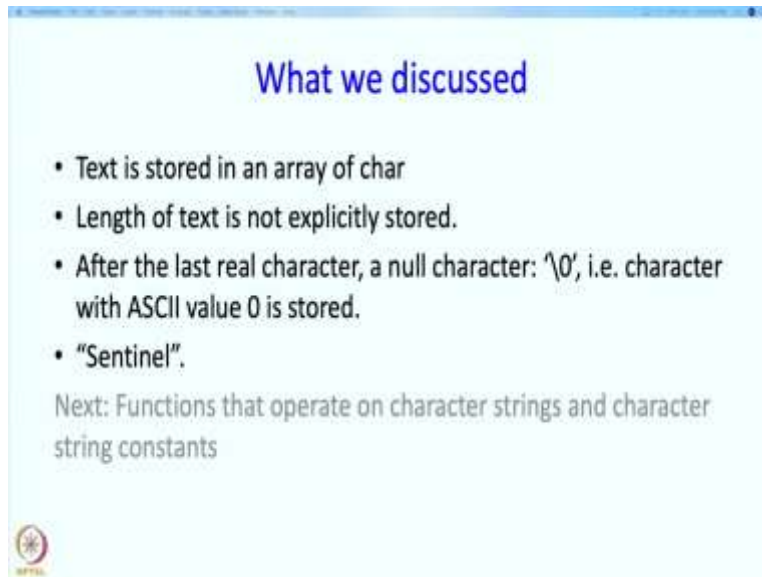


**An Introduction to Programming through C++**  
**Professor Abhiram G. Ranade**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology Bombay**  
**Lecture No. 17 Part - 2**  
**More on Arrays**  
**Functions on character strings**


(Refer Slide Time: 0:22)



**What we discussed**

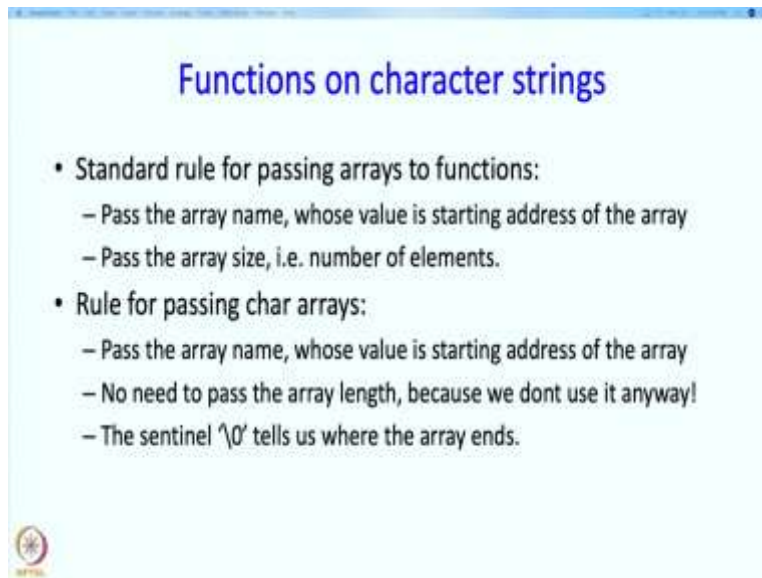
- Text is stored in an array of char
- Length of text is not explicitly stored.
- After the last real character, a null character: `'\0'`, i.e. character with ASCII value 0 is stored.
- "Sentinel".

Next: Functions that operate on character strings and character string constants



Welcome back, in the last segment we discussed how text is stored on a computer and in this, in this segment we are going to talk about functions that operate on character strings and also the notion of character string constants.

(Refer Slide Time: 0:42)



The slide is titled "Functions on character strings" in blue text. It contains two main bullet points. The first is "Standard rule for passing arrays to functions:" with two sub-points: "Pass the array name, whose value is starting address of the array" and "Pass the array size, i.e. number of elements." The second is "Rule for passing char arrays:" with three sub-points: "Pass the array name, whose value is starting address of the array", "No need to pass the array length, because we dont use it anyway!", and "The sentinel '\0' tells us where the array ends." There is a small logo in the bottom left corner of the slide.

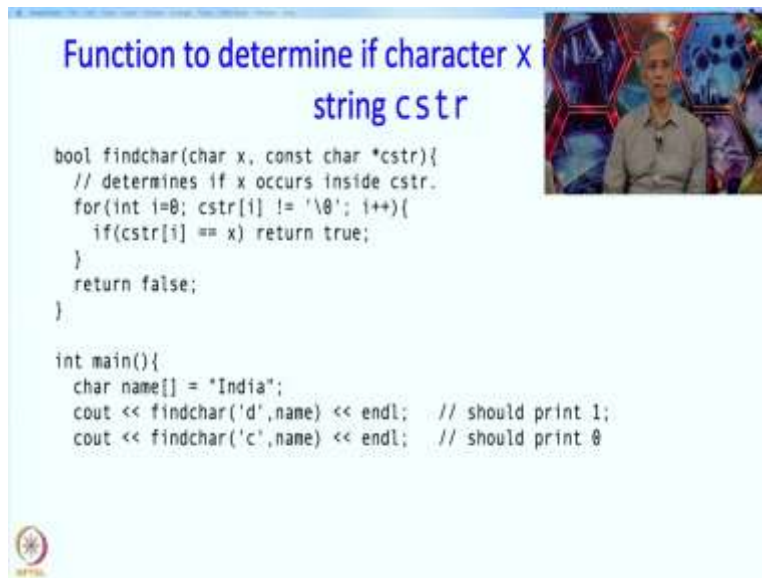
- Standard rule for passing arrays to functions:
  - Pass the array name, whose value is starting address of the array
  - Pass the array size, i.e. number of elements.
- Rule for passing char arrays:
  - Pass the array name, whose value is starting address of the array
  - No need to pass the array length, because we dont use it anyway!
  - The sentinel '\0' tells us where the array ends.

So there is a standard rule for passing arrays to functions, so what is that rule? You pass the array name and the value of the array name is simply the starting address of the array, so when you pass the array name you are telling the function where the array is beginning in memory. And then you pass the array size, so using these two things the function can figure out how many elements the array has, and then the function inside the function also you can do the address calculation to get the different elements.

And the array size, well you can check, you can look at the array size to decide what is reasonable or what is a legal value of the index. The rule for passing char arrays to functions is slightly different. So we have to pass the array name, so the function has to be told where the array starts in memory, from which address the array is allocated. But, there is no need to pass the array length, because of the of our convention, we only process till the null character, so we do not really worry about the length of the array in any case. So therefore, there is no need to pass it.

So again, to reiterate we do not look at the length of the array, but we keep processing until we encounter the sentinel, which is the null character which is the character whose ASCII value is 0, and that tells us that oh the array has ended over here.

(Refer Slide Time: 2:17)



### Function to determine if character x string cstr

```
bool findchar(char x, const char *cstr){
    // determines if x occurs inside cstr.
    for(int i=0; cstr[i] != '\0'; i++){
        if(cstr[i] == x) return true;
    }
    return false;
}

int main(){
    char name[] = "India";
    cout << findchar('d',name) << endl; // should print 1;
    cout << findchar('c',name) << endl; // should print 0
```

So, let us write a simple function for on strings, so this is a function which determines if a character 'x' is present in a string cstr. So it takes as argument the character x itself and character array cstr, so we are going to call it constant because inside this that array is that cstr is not changed. As said in the title the purpose of this function is to determine if x occurs inside cstr and so it will return a true if x occurs and return a false if it does not matter, so how does this work? So we are going to start scanning the cstr from index 0, so i is going to be our index and so we are going to write int i=0, and condition now is until we encounter the null character. So the for loop has this condition and often the condition deals with the length of the array, but in this case we will just say that look i, the element at index i should not be a null character and then the update is the same as before, so we increment i, we go to the next index or we scan forward.

And here, if we encounter x, if at any index we encounter x, then the return true, if we go through this entire loop and we do not encounter x that is when we will come out, our control will come to this point. So if control reaches this point, then that means x does not appear in this string cstr, and so we return false. So here is how this program could be called through, could be, a function could be called through a main program, so we are going to set a character array to "India" and then we can say findchar(d) in the name.

So in this case d does appear in the string India or in the string stored in name and therefore, they should print, this will print a 1. As you may remember true is not printed, but 1 is printed instead. And, if you try say to search for a c, it will not be found and therefore, we will print a 0.

(Refer Slide Time: 5:06)



Demo

- findchar.cpp

```
File Edit Options Buffers Tools C++ Help
#include <string>
using namespace std;

bool findchar(char x, const char *cstr){
    // determines if x occurs inside cstr.
    for(int i=0; cstr[i] != '\0'; i++){
        if(cstr[i] == x) return true;
    }
    return false;
}

int main(){
    char name[] = "India";
    cout << findchar('d',name) << endl; // should print 1;
    cout << findchar('c',name) << endl; // should print 0
}

-UU-r-----F1 findchar.cpp All LI (C++/l Abbrev) 3:18PM 0.91
```

```

0JVM4nHijpJnwsTNwK0ixqX3L4LYxNb1k3MeH3MxuCsWnDCUhlVASJbc3L8wLXDP9RWwmxT0Zc
/cyagVxk3dy/NBcgdg50pjnd83ZGbg9H9K7Zg7Zcg3Ndrho1h8xwE0AJ03Fe058WwAGJY1K4L
fii478fEgbLX4UzW5t1S+IfwZs1QJmfnd00JUZpjnw9xQp80S2KcQ5moaXTtEyH9zJ1JY1J1z
jmU73EBpv7gpYUGfUty3ipjnH++J1XP9RzctceZgpz/ol9kEa3WpQIyz/qA2WwE1jLueWV+YzWg1
~) %
emacs -nw
~) logout
Connection to surya.cse.iitb.ac.in closed.
~/Desktop/nptel/week8 : ls
Bsearch.cpp  Lec8.2.pptx  findchar.cpp~  mergesort.cpp
Bsearch.cpp~  a.out*      house.cpp      mergesort.cpp~
Lec8.1.pptx  findchar.cpp  house.cpp~
~/Desktop/nptel/week8 :
~/Desktop/nptel/week8 :
~/Desktop/nptel/week8 : open Lec8.1.pptx
~/Desktop/nptel/week8 : %
emacs -nw highest.cpp  (wd: ~/Desktop/nptel/week7)

[1]+  Stopped                  emacs -nw highest.cpp  (wd: ~/Desktop/nptel/week7)
(wd now: ~/Desktop/nptel/week8)
~/Desktop/nptel/week8 : s++ findchar.cpp
+ g++ findchar.cpp -Wall -I/Users/abhiram/simplecpp/lib/libspriete.a -I/Users/abhiram/simplecpp
-I/opt/X11/include -L/opt/X11/lib -lX11 -std=c++17
~/Desktop/nptel/week8 : ./a.out
1
0
~/Desktop/nptel/week8 :

```

So let us do a quick demo of this, so here is the file. So I am going to compile it now, and then run it, so as expected 1 was printed followed by 0. So here is another piece of code which is very similar and I want to ask you whether you think this should also work?

(Refer Slide Time: 5:42)

**Shouldn't this also work?**

```

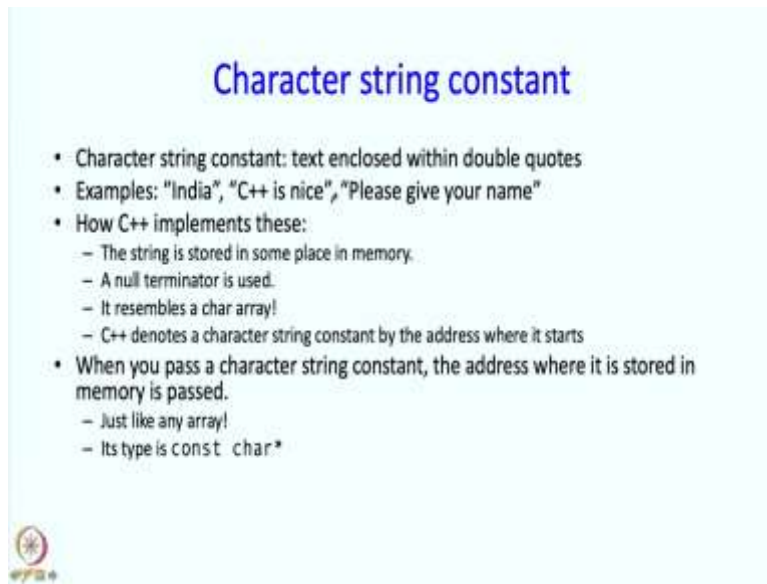
bool findchar(char x, const char *cstr){
    // determines if x occurs inside cstr.
    // Note that length of array cstr is not specified - not needed.
    for(int i=0; cstr[i] != '\0'; i++){
        if(cstr[i] == x) return true;
    }
    return false;
}

int main(){
    cout << findchar('d',"India") << endl; // should print 1;
    cout << findchar('c',"India") << endl; // should print 0
}
• it does, but how?

```


So in the previous program what we did was we put a string to an array and then called 'findchar', but you would think or you would desire that something like this should also work, why should we have to put that string into an array and then call it, why cannot we send this string off directly? And similarly, if we try to find 'c' in "India" that should print to 0, well this does work and what we are going to do next is to see how or why this works.

(Refer Slide Time: 6:33)



### Character string constant

- Character string constant: text enclosed within double quotes
- Examples: "India", "C++ is nice", "Please give your name"
- How C++ implements these:
  - The string is stored in some place in memory.
  - A null terminator is used.
  - It resembles a char array!
  - C++ denotes a character string constant by the address where it starts
- When you pass a character string constant, the address where it is stored in memory is passed.
  - Just like any array!
  - Its type is `const char*`



So for this we need the notion of a character string constant. Character string constant, constants are simply text enclosed within double quotes. For example, "India" is a character string constant, "C++ is nice" and "please give your name" are also character string constants, they are constants in the sense that they are not variables, you cannot do, you cannot change things inside this, you are not expected to change things inside this.

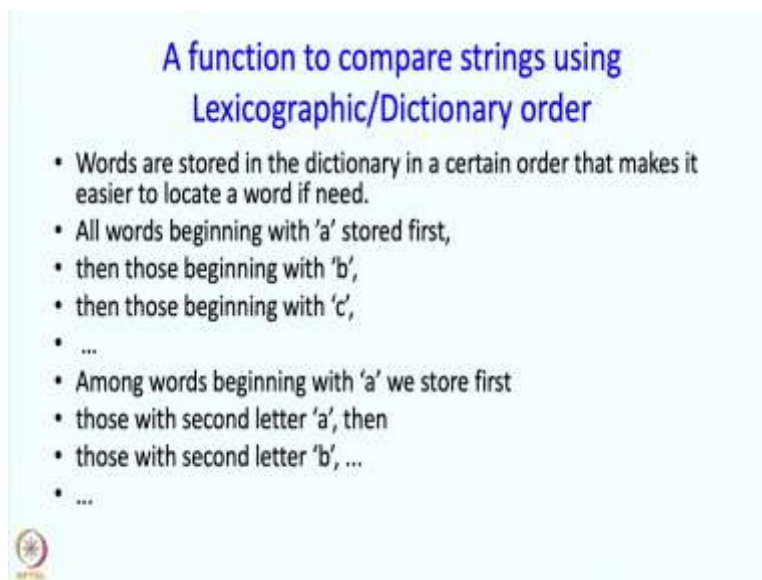
Now, how does C++ deal with such constants? Well first of all C++ stores the string in some place in memory and of course a null terminator is also stored, so that is the convention and sort of it is useful to follow conventions sort of, sort of blindly. So the null terminator is used and you will see that it comes in handy. And now this resembles char array, so C++ denotes character string constant by the address where it starts. So it really treats it like a char array, but it is a char array that C++ itself has created.

So when you, when you type this C++ looks at it and it stores it somewhere and it says that oh look whatever you have typed here is at this address and it is the array stored at this address. So when you pass a character string constant, the address where you stored it memory is passed, so this happens just like any array. And I should note that the type of this is `const char*`, so what that means is, you cannot modify the content of this inside, inside the function, rather I should say you will not modify, you are promising that you will not modify the array that is being passed.

And indeed C++ does not want you to do that, so if you send, if you write, if you send this as an argument to, say, `findchar`, then C++ does not want you to change the value of this “C++ is nice” this entire thing. There are reasons for it, C++ may store it in regions of memory which the computer may have some special restrictions on, so that when the user program executes, that region of memory is read only, that that may will be possible.


So as a result C++ says that look you are not supposed to change this and the way C++ makes sure that does not happen is by requiring you to declare this argument as `const`. In any case you do not, you are not intending to modify that argument when you try to find a character in some string, you are not intending to modify that string, anyway so there is no harm for you in declaring it as a `const`.

(Refer Slide Time: 9:41)



A function to compare strings using  
Lexicographic/Dictionary order

- Words are stored in the dictionary in a certain order that makes it easier to locate a word if need.
- All words beginning with 'a' stored first,
- then those beginning with 'b',
- then those beginning with 'c',
- ...
- Among words beginning with 'a' we store first
- those with second letter 'a', then
- those with second letter 'b', ...
- ...

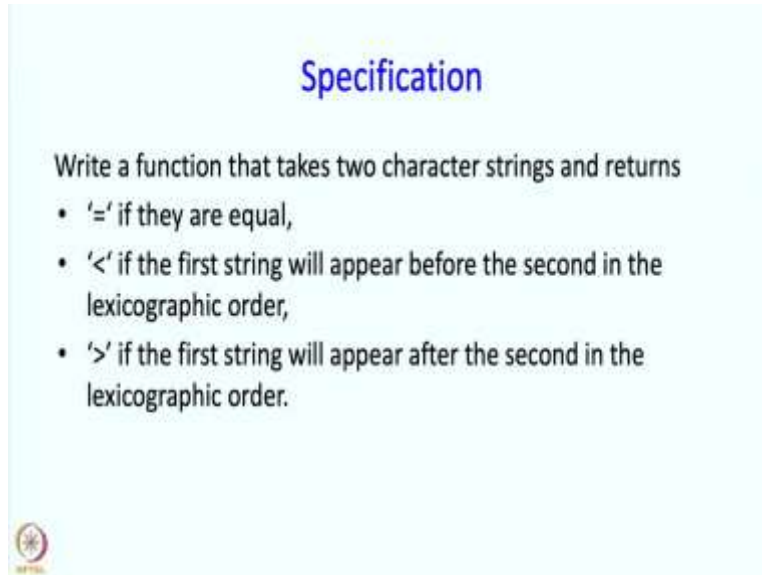


So that was relatively simple function and that was a discussion of what are character strings. Now, we move on to a little bit more complicated function, but more than the complexity, it is a new notion that we want to introduce, this is a notion that is very commonly used with character strings, that is comparing them using the lexicographic or the dictionary order. So, what is that? Well words are stored in the dictionary in a certain order that makes it easier to locate the word if needed, you know this when you search a dictionary, what words come first?

Well all the words beginning with 'a' come first, then those beginning with 'b', then those beginning with 'c' and so on. And within the words that begin with 'a' what is stored first? Well

those with second letter 'a', then those with second letter 'b' and so on. So this is the so-called dictionary order of words or the lexicographic order of words.


(Refer Slide Time: 10:56)



### Specification

Write a function that takes two character strings and returns

- '=' if they are equal,
- '<' if the first string will appear before the second in the lexicographic order,
- '>' if the first string will appear after the second in the lexicographic order.



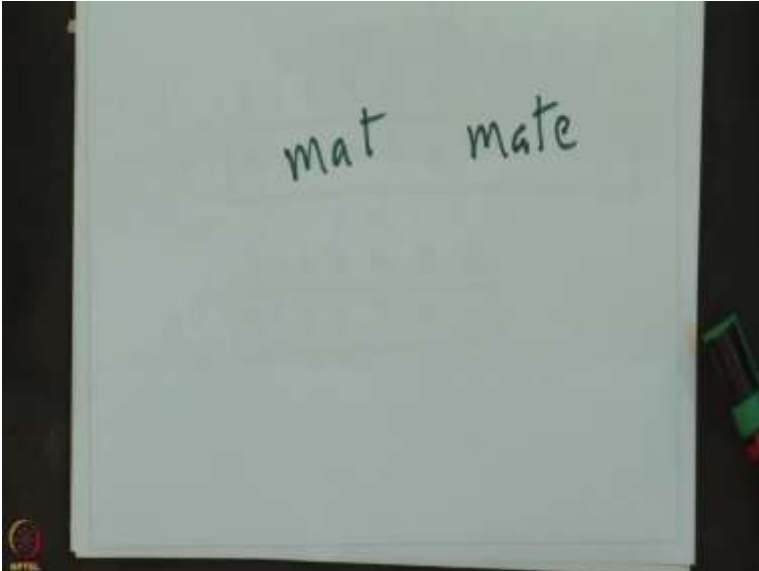
So, the function that we are going to write has the following specification. So it takes two character strings and returns '=' if the strings are exactly equal, so it returns the character '=', it returns the character '<' if the first string will appear before the second in the lexicographic order. And it will return '>' if the first string will appear after the second in the lexicographic order. So this is an order, this is an order between words and you can say that something appears before so it is smaller, so that is also an implied notion, or it is more like an associated notion.



(Refer Slide Time: 11:44)

### Solution

```
char compare(const char *a, const char *b)
/*
Start from the beginning of both and compare corresponding
letters.
If they are same move on to next letter of both.
If at any point you find a's letter smaller, then return '<'.
Likewise if b's letter.
If both strings end without finding a difference return '='.
If one string ends, its letter will be '\0' which is smaller
than any other letter, so this case will be dealt with above.
*/
```



So, how do you write this? Well, here is the function. So let us call it compare, and it is going to return a char as we said and it should take two character strings as arguments, so the two that we want to compare. And since we do not intend to modify those two strings, we are going to make them constant. And of course, if we make them constant then we can also supply as argument character string constants if we wish.

So, what is this supposed to do? Well, we will start from the beginning of both, and compare corresponding letters. If they are the same, then, so far two strings look equal, you do not know which one will be stored before in the dictionary and which one will be stored later. So you have

to move on to the next letter of both. If at any point you find that a's letter is smaller than the letter of b, then what does that mean? That means that 'a' will be stored before in the dictionary order and so you can immediately return 'less than'.


Likewise, if you find that b's letter is smaller, then you can immediately return 'greater than'. If both strings end without finding a, without there being a difference at any position, then you can just return '='. And if one string ends, its letter will be null and null is ASCII 0, and it is smaller than any other letter. So effectively this case if you are comparing the two, the corresponding letters, then the letter which is ending will be considered, the letter will be smaller. And therefore, by the previous logic queue will be returning '<'.

But that is good because the letters which terminate, the words which terminate earlier are considered to be coming first. So, for example, if I write mat that appears before mate. So that is exactly, so that will also be handled properly by our mechanism over here. So that is really what the algorithm is.

(Refer Slide Time: 14:28)

```
Solution (contd.)
```

```
{
  int i=0;
  while(true){
    if(a[i]=='\0' && b[i]=='\0')
      return '=';
    if(a[i]<b[i]) return '<';
    if(a[i]>b[i]) return '>';
    i++;
  }
} // more compact than the book.
```

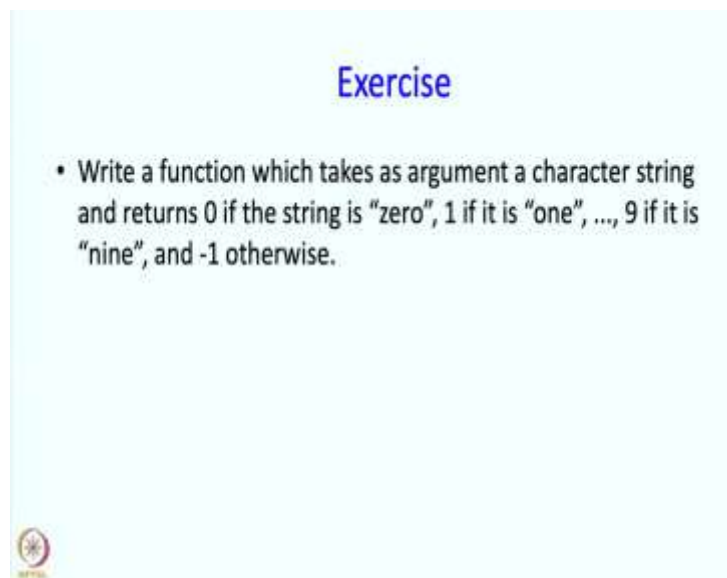


And so here is the solution, we start with i=0, this is the position from which we compare. And while, while true or we are getting into, we are not, we are always going to the next iteration over here. So, if we find that both the ith element of 'a' as well as the ith element of 'b' are null, then that means we have gone through the loop several times and we have not discovered a difference

and we have come to the end of both words. And so we can return equality, we have discovered that those words are equal.

If we find that  $a[i] < b[i]$ , then as we said we can return less than, if we find  $a[i] > b[i]$  then we can return greater than. And if none of these is true, so what is that case? So that case is if  $a[i] == b[i]$ , then we have to go and examine the next letter and so we do  $i++$ , so this code is also discussed in the book, but this is slightly more compact.

(Refer Slide Time: 15:46)

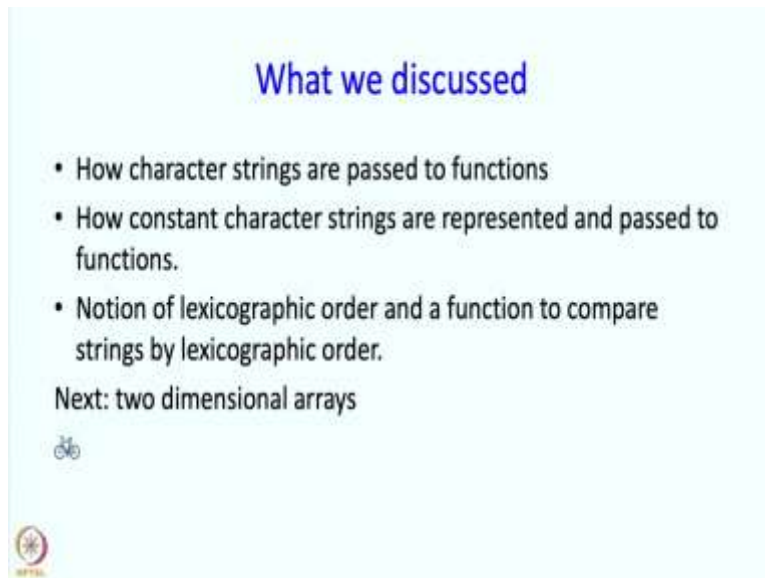


### Exercise

- Write a function which takes as argument a character string and returns 0 if the string is "zero", 1 if it is "one", ..., 9 if it is "nine", and -1 otherwise.

So you have to, as an exercise you have to write a function which takes as argument a character string and returns 0, if the string is 0, 1 if it is 1, and so on, and 9 if it is 9, and minus 1 otherwise. So, you can just compare say the given text string with these text strings and depending upon what the comparison is, you can just let return the appropriate numeral.


(Refer Slide Time: 16:17)




**What we discussed**

- How character strings are passed to functions
- How constant character strings are represented and passed to functions.
- Notion of lexicographic order and a function to compare strings by lexicographic order.

Next: two dimensional arrays





Alright, so what have we discussed? So we have talked about how character strings are passed to functions, then we also talked about how constant character strings are represented, and how they are passed to functions? Basically you put a const, then we talked about the notion of lexicographic order, and we discussed a function to compare strings by the lexicographic order. We will take a break now and in the next segment we will discuss two-dimensional arrays.